Instruction scheduling

Advanced Compiler Construction
Michel Schinz – 2013-05-16

Instruction ordering

When a compiler emits the instructions corresponding to a program, it imposes a total order on them.
However, that order is usually not the only valid one, in the sense that it can be changed without modifying the program's behavior.
For example, if two instructions i_1 and i_2 appear sequentially in that order and are independent, then it is possible to swap them.

Instruction scheduling

Among all the valid permutations of the instructions composing a program – i.e. those that preserve the program's behavior – some can be more desirable than others. For example, one order might lead to a faster program on some machine, because of architectural constraints.
The aim of **instruction scheduling** is to find a valid order that optimizes some metric, like execution speed.

Pipeline stalls

Modern, pipelined architectures can usually issue at least one instruction per clock cycle.
However, an instruction can be executed only if the data it needs is ready. Otherwise, the pipeline **stalls** for one or several cycles.
Stalls can appear because some instructions – e.g. division – require several cycles to complete, or because data has to be fetched from memory.

Scheduling example

The following example will illustrate how proper scheduling can reduce the time required to execute a piece of RTL code.
We assume the following delays for instructions:

Instruction kind	RTL notation	Delay
Memory load or store	$R_a \leftarrow \text{Mem}[R_b+c]$ $\text{Mem}[R_b+c] \leftarrow R_a$	3
Multiplication	$R_a \leftarrow R_b * R_c$	2
Addition	$R_a \leftarrow R_b + R_c$	1

Scheduling example

Cycle	Instruction
1	$R_1 \leftarrow \text{Mem}[R_{SP}]$
4	$R_1 \leftarrow R_1 + R_1$
5	$R_2 \leftarrow \text{Mem}[R_{SP}+1]$
8	$R_1 \leftarrow R_1 * R_2$
9	$R_2 \leftarrow \text{Mem}[R_{SP}+2]$
12	$R_1 \leftarrow R_1 * R_2$
13	$R_2 \leftarrow \text{Mem}[R_{SP}+3]$
16	$R_1 \leftarrow R_1 * R_2$
18	$\text{Mem}[R_{SP}+4] \leftarrow R_1$

Cycle	Instruction
1	$R_1 \leftarrow \text{Mem}[R_{SP}]$
2	$R_2 \leftarrow \text{Mem}[R_{SP}+1]$
3	$R_3 \leftarrow \text{Mem}[R_{SP}+2]$
4	$R_1 \leftarrow R_1 + R_1$
5	$R_1 \leftarrow R_1 * R_2$
6	$R_2 \leftarrow \text{Mem}[R_{SP}+3]$
7	$R_1 \leftarrow R_1 * R_3$
9	$R_1 \leftarrow R_1 * R_2$
11	$\text{Mem}[R_{SP}+4] \leftarrow R_1$

After scheduling (including renaming), the last instruction is issued at cycle 11 instead of 18!

Instruction dependences

An instruction i_2 **depends** on an instruction i_1 when it is not possible to execute i_2 before i_1 without changing the behavior of the program.
The most common reason for dependence is data-dependence: i_2 uses a value that is computed by i_1.
However, as we will see, there are other kinds of dependences.

Data dependences

We distinguish three kinds of dependences between two instructions i_1 and i_2:

- true dependence – i_2 reads a value written by i_1 (**read after write**, **RAW**),
- antidependence – i_2 writes a value read by i_1 (**write after read**, **WAR**),
- antidependence – i_2 writes a value written by i_1 (**write after write**, **WAW**).

Antidependences

Antidependences are not real dependences, in the sense that they do not arise from the flow of data. They are due to a single location being used to store different values.

Most of the time, antidependences can be removed by renaming locations – e.g. registers.

In the example below, the program on the left contains a WAW antidependence between the two memory load instructions, that can be removed by renaming the second use of R_1.

```
R1 ← Mem[RSP]                R1 ← Mem[RSP]
R4 ← R4 + R1                 R4 ← R4 + R1
R1 ← Mem[RSP+1]      ➡       R2 ← Mem[RSP+1]
R4 ← R4 + R1                 R4 ← R4 + R2
```

Computing dependences

Identifying dependences among instructions that only access registers is easy.

Instructions that access memory are harder to handle. In general, it is not possible to know whether two such instructions refer to the same memory location.

Conservative approximations – not examined here – therefore have to be used.

Dependence graph

The **dependence graph** is a directed graph representing dependences among instructions.

Its nodes are the instructions to schedule, and there is an edge from node n_1 to node n_2 iff the instruction of n_2 depends on n_1.

Any topological sort of the nodes of this graph represents a valid way to schedule the instructions.

Dependence graph example

Name	Instruction
a	`R1 ← Mem[RSP]`
b	`R1 ← R1 + R1`
c	`R2 ← Mem[RSP+1]`
d	`R1 ← R1 * R2`
e	`R2 ← Mem[RSP+2]`
f	`R1 ← R1 * R2`
g	`R2 ← Mem[RSP+3]`
h	`R1 ← R1 * R2`
i	`Mem[RSP+4] ← R1`

$$\begin{tikzcd}
 & a \arrow[dl] & & & \\
b \arrow[dr] & & c \arrow[dl] \arrow[dr, dotted] & & \\
 & d \arrow[rr, dotted] \arrow[dr] & & e \arrow[dl] \arrow[dr, dotted] & \\
 & & f \arrow[rr, dotted] \arrow[dr] & & g \arrow[dl] \\
 & & & h \arrow[dr] & \\
 & & & & i
\end{tikzcd}$$

→ true dependence

⋯▸ antidependence

Difficulty of scheduling

Optimal instruction scheduling is NP-complete.
As always, this implies that we will use techniques based on heuristics to find a good – but sometimes not optimal – solution to that problem.
List scheduling is a technique to schedule the instructions of a single basic block.
Its basic idea is to simulate the execution of the instructions, and to try to schedule instructions only when all their operands can be used without stalling the pipeline.

List scheduling algorithm

The list scheduling algorithm maintains two lists:

- `ready` is the list of instructions that could be scheduled without stall, ordered by priority,
- `active` is the list of instructions that are being executed.

At each step, the highest-priority instruction from `ready` is scheduled, and moved to `active`, where it stays for a time equal to its delay.
Before scheduling is performed, renaming is done to remove all antidependences that can be removed.

Prioritizing instructions

Nodes (i.e. instructions) are sorted by priority in the `ready` list. Several schemes exist to compute the priority of a node, which can be equal to:

- the length of the longest latency-weighted path from it to a root of the dependence graph,
- the number of its immediate successors,
- the number of its descendants,
- its latency,
- etc.

Unfortunately, no single scheme is better for all cases.

List scheduling example

$$\begin{tikzcd}
& a^{13} \arrow[dl] & & & \\
b^{10} \arrow[dr] & & c^{12} \arrow[dl] & & \\
& d^{9} \arrow[dr] & & e^{10} \arrow[dl] & \\
& & f^{7} \arrow[dr] & & g^{8} \arrow[dl] \\
& & & h^{5} \arrow[dr] & \\
& & & & i^{3}
\end{tikzcd}$$

A node's priority is the length of the longest latency-weighted path from it to a root of the dependence graph

Cycle	ready	active
1	$[a^{13},c^{12},e^{10},g^{8}]$	[a]
2	$[c^{12},e^{10},g^{8}]$	[a,c]
3	$[e^{10},g^{8}]$	[a,c,e]
4	$[b^{10},g^{8}]$	[b,c,e]
5	$[d^{9},g^{8}]$	[d,e]
6	$[g^{8}]$	[d,g]
7	$[f^{7}]$	[f,g]
8	[]	[f,g]
9	$[h^{5}]$	[h]
10	[]	[h]
11	$[i^{3}]$	[i]
12	[]	[i]
13	[]	[i]
14	[]	[]

Scheduling conflicts

It is hard to decide whether scheduling should be done before or after register allocation.
If register allocation is done first, it can introduce antidependences when reusing registers.
If scheduling is done first, register allocation can introduce spilling code, destroying the schedule.
Solution: schedule first, then allocate registers and schedule once more if spilling was necessary.